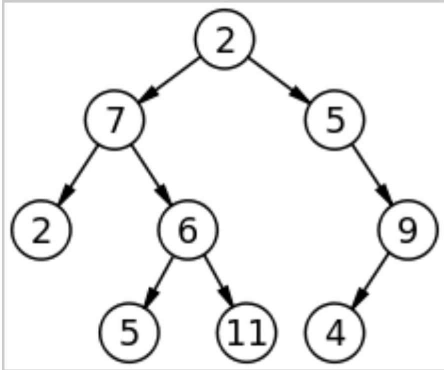
Tree (data structure)

From Wikipedia, the free encyclopedia

In computer science, a **tree** is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.



A simple unordered tree; in this diagram, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.

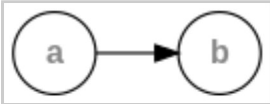
A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

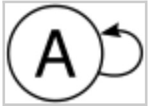
Contents

Definition

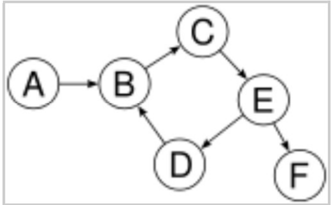
A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.



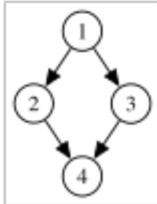
Each linear list is trivially a tree



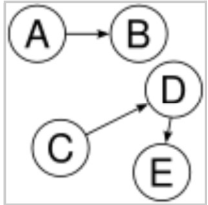
Not a tree: cycle A→A. A is the root but it also has a parent.



Not a tree: cycle B→C→E→D→B. B has more than one parent (inbound edge).



Not a tree: undirected cycle 1-2-4-3. 4 has more than one parent (inbound edge).



Not a tree: two non-connected parts, A→B and C→D→E. There is more than one root.

Terminology used in trees

Root
: The top node in a tree.

Child
: A node directly connected to another node when moving away from the Root.

Parent
: The converse notion of a *child*.

Siblings
: A group of nodes with the same parent.

Descendant
: A node reachable by repeated proceeding from parent to child.

Ancestor

A node reachable by repeated proceeding from child to parent.

Leaf
(less commonly called External node)
A node with no children.

Branch
Internal node
A node with at least one child.

Degree
The number of sub trees of a node.

Edge
The connection between one node and another.

Path
A sequence of nodes and edges connecting a node with a descendant.

Level
The level of a node is defined by 1 + (the number of connections between the node and the root).

Height of node
The height of a node is the number of edges on the longest path between that node and a leaf.

Height of tree
The height of a tree is the height of its root node.

Depth
The depth of a node is the number of edges from the tree's root node to the node.

Forest
A forest is a set of $n \geq 0$ disjoint trees.

Data type vs. data structure

There is a distinction between a tree as an abstract data type and as a concrete data structure, analogous to the distinction between a list and a linked list. As a data type, a tree has a value and children, and the children are themselves trees; the value and children of the tree are interpreted as the value of the root node and the subtrees of the children of the root node. To allow finite trees, one must either allow the list of children to be empty (in which case trees can be required to be non-empty, an "empty tree" instead being represented by a forest of zero trees), or allow trees to be empty, in which case the list of children can be of fixed size (branching factor, especially 2 or "binary"), if desired.

As a data structure, a linked tree is a group of nodes, where each node has a value and a list of references to other nodes (its children). This data structure actually defines a directed graph,[a] because it may have loops or several references to the same node, just as a linked list may have a loop. Thus there is also the requirement that no two references point to the same node (that each node has at most a single parent, and in fact exactly one parent, except for the root), and a tree that violates this is "corrupt".

Due to the use of *references* to trees in the linked tree data structure, trees are often discussed implicitly assuming that they are being represented by references to the root node, as this is often how they are actually implemented. For example, rather than an empty tree, one may have a null reference: a tree is always non-empty, but a reference to a tree may be null.

Recursive

Recursively, as a data type a tree is defined as a value (of some data type, possibly empty), together with a list of trees (possibly an empty list), the subtrees of its children; symbolically:

```
t: v [t[1], ..., t[k]]
```

(A tree *t* consists of a value *v* and a list of other trees.)

More elegantly, via mutual recursion, of which a tree is one of the most basic examples, a tree can be defined in terms of a forest (a list of trees), where a tree consists of a value and a forest (the subtrees of its children):

```
f: [t[1], ..., t[k]]
t: v f
```

Note that this definition is in terms of values, and is appropriate in functional languages (it assumes referential transparency); different trees have no connections, as they are simply lists of values.

As a data structure, a tree is defined as a node (the root), which itself consists of a value (of some data type, possibly empty), together with a list of references to other nodes (list possibly empty, references possibly null); symbolically:

```
n: v [&n[1], ..., &n[k]]
```

(A node *n* consists of a value *v* and a list of references to other nodes.)

This data structure defines a directed graph,[b] and for it to be a tree one must add a condition on its global structure (its topology), namely that at most one reference can point to any given node (a node has at most a single parent), and no node in the tree point to the root. In fact, every node (other than the root) must have exactly one parent, and the root must have no parents.

Indeed, given a list of nodes, and for each node a list of references to its children, one cannot tell if this structure is a tree or not without analyzing its global structure and that it is in fact topologically a tree, as defined below.

Type theory

As an ADT, the abstract tree type T with values of some type E is defined, using the abstract forest type F (list of trees), by the functions:

value: $T \to E$
children: $T \to F$
nil: $() \to F$
node: $E \times F \to T$

with the axioms:

value(node(e, f)) = e
children(node(e, f)) = f

In terms of type theory, a tree is an inductive type defined by the constructors *nil* (empty forest) and *node* (tree with root node with given value and children).

Mathematical

Viewed as a whole, a tree data structure is an ordered tree, generally with values attached to each node. Concretely, it is (if required to be non-empty):

- A rooted tree with the "away from root" direction (a more narrow term is an "arborescence"), meaning:
  - A directed graph,
  - whose underlying undirected graph is a tree (any two vertices are connected by exactly one simple path),
  - with a distinguished root (one vertex is designated as the root),
  - which determines the direction on the edges (arrows point away from the root; given an edge, the node that the edge points from is called the *parent* and the node that the edge points to is called the *child*),

together with:

- an ordering on the child nodes of a given node, and

- a value (of some data type) at each node.

Often trees have a fixed (more properly, bounded) branching factor (outdegree), particularly always having two child nodes (possibly empty, hence *at most* two *non-empty* child nodes), hence a "binary tree".

Allowing empty trees makes some definitions simpler, some more complicated: a rooted tree must be non-empty, hence if empty trees are allowed the above definition instead becomes "an empty tree, or a rooted tree such that ...". On the other hand, empty trees simplify defining fixed branching factor: with empty trees allowed, a binary tree is a tree such that every node has exactly two children, each of which is a tree (possibly empty).The complete sets of operations on tree must include fork operation.

Terminology

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes**, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's **parent node** (or *ancestor node*, or superior). A node has at most one parent.

An **internal node** (also known as an **inner node**, **inode** for short, or **branch node**) is any node of a tree that has child nodes. Similarly, an **external node** (also known as an **outer node**, **leaf node**, or **terminal node**) is any node that does not have child nodes.

The topmost node in a tree is called the **root node**. Depending on definition, a tree may be required to have a root node (in which case all trees are non-empty), or may be allowed to be empty, in which case it does not necessarily have a root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children. Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the *value* of the root last). All other nodes can be reached from it by following **edges** or **links**. (In the formal definition, each such path is also unique.) In diagrams, the root node is conventionally drawn at the top. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its *root path*). This is commonly needed in the manipulation of the various self-balancing trees, AVL Trees in particular. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has depth and height -1.

A **subtree** of a tree T is a tree consisting of a node in T and all of its descendants in T.[c][1] Nodes thus correspond to subtrees (each node corresponds to the subtree of itself and all its descendants) – the subtree corresponding to the root node is the entire tree, and each node is the root node of the subtree it determines; the subtree corresponding to any other node is called a **proper subtree** (by analogy to a proper subset).

Drawing trees

Trees are often drawn in the plane. Ordered trees can be represented essentially uniquely in the plane, and are hence called *plane trees,* as follows: if one fixes a conventional order (say, counterclockwise), and arranges the child nodes in that order (first incoming parent edge, then first child edge, etc.), this yields an embedding of the tree in the plane, unique up to ambient isotopy. Conversely, such an embedding determines an ordering of the child nodes.

If one places the root at the top (parents above children, as in a family tree) and places all nodes that are a given distance from the root (in terms of number of edges: the "level" of a tree) on a given horizontal line, one obtains a standard drawing of the tree. Given a binary tree, the first child is on the left (the "left node"), and the second child is on the right (the "right node").

Representations

There are many different ways to represent trees; common representations represent the nodes as dynamically allocated records with pointers to their children, their parents, or both, or as items in an array, with relationships between them determined by their positions in the array (e.g., binary heap).

Indeed, a binary tree can be implemented as a list of lists (a list where the values are lists): the head of a list (the value of the first term) is the left child (subtree), while the tail (the list of second and subsequent terms) is the right child (subtree). This can be modified to allow values as well, as in Lisp S-expressions, where the head (value of first term) is the value of the node, the head of the tail (value of second term) is the left child, and the tail of the tail (list of third and subsequent terms) is the right child.

In general a node in a tree will not have pointers to its parents, but this information can be included (expanding the data structure to also include a pointer to the parent) or stored separately. Alternatively, upward links can be included in the child node data, as in a threaded binary tree.

Generalizations

Digraphs

If edges (to child nodes) are thought of as references, then a tree is a special case of a digraph, and the tree data structure can be generalized to represent directed graphs by removing the constraints that a node may have at most one parent, and that no cycles are allowed. Edges are still abstractly considered as pairs of nodes, however, the terms *parent* and *child* are usually replaced by different terminology (for example, *source* and *target*). Different implementation strategies exist: a digraph can be represented by the same local data structure as a tree (node with value and list of children), assuming that "list of children" is a list of references, or globally by such structures as adjacency lists.

In graph theory, a tree is a connected acyclic graph; unless stated otherwise, in graph theory trees and graphs are assumed undirected. There is no one-to-one correspondence between such trees and trees as data structure. We can take an arbitrary undirected tree, arbitrarily pick one of its vertices as the *root*, make all its edges directed by making them point away from the root node – producing an arborescence – and assign an order to all the nodes. The result corresponds to a tree data structure. Picking a different root or different ordering produces a different one.

Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root). Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for corecursion (as in a breadth-first search).

Via mutual recursion, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children):

```
f: [n[1], ..., n[k]]
n: v f
```

Traversal methods

Stepping through the items of a tree, by means of the connections between parents and children, is called **walking the tree**, and the action is a **walk** of the tree. Often, an operation might be performed when a pointer arrives at a particular node. A walk in which each parent node is traversed before its children is called a **pre-order** walk; a walk in which the children are traversed before their respective parents are traversed is called a **post-order** walk; a walk in which a node's left subtree, then the node itself, and finally its right subtree are traversed is called an **in-order** traversal. (This last scenario, referring to exactly two subtrees, a left subtree and a right subtree, assumes specifically a binary tree.) A **level-order** walk effectively performs a breadth-first search over the entirety of a tree; nodes are traversed level by level, where the root node is visited first, followed by its direct child nodes and their siblings, followed by its grandchild nodes and their siblings, etc., until all nodes in the tree have been traversed.

Common operations

- Enumerating all the items
- Enumerating a section of a tree
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Pruning: Removing a whole section of a tree
- Grafting: Adding a whole section to a tree
- Finding the root for any node

Common uses

- Representing hierarchical data
- Storing data in a way that makes it efficiently searchable (see binary search tree and tree traversal)
- Representing sorted lists of data
- As a workflow for compositing digital images for visual effects
- Routing algorithms

See also

- Tree structure
- Tree (graph theory)
- Tree (set theory)
- Hierarchy (mathematics)
- Dialog tree
- Single inheritance
- Generative grammar
- Hierarchical clustering
- Binary space partition tree
- Recursion

Other trees

- Trie
- DSW algorithm
- Enfilade
- Left child-right sibling binary tree
- Hierarchical temporal memory

Notes

a. Properly, a rooted, ordered directed graph.
b. Properly, a rooted, ordered directed graph.
c. This is different from the formal definition of subtree used in graph theory, which is a subgraph that forms a tree – it need not include all descendants. For example, the root node by itself is a subtree in the graph theory sense, but not in the data structure sense (unless there are no descendants).

References

1. Weisstein, Eric W. "Subtree". *MathWorld*.

- Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Section 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

External links

Wikimedia Commons has media related to ***Tree structures***.

- Data Trees as a Means of Presenting Complex Data Analysis (http://www.community-of-knowledge.de/beitrag/data-trees-as-a-means-of-presenting-complex-data-analysis/) by Sally Knipe
- Description (https://xlinux.nist.gov/dads/HTML/tree.html) from the Dictionary of Algorithms and Data Structures
- data.tree (http://www.ipub.com/data.tree) implementation of a tree data structure in the R programming language
- WormWeb.org: Interactive Visualization of the *C. elegans* Cell Tree (http://wormweb.org/celllineage) – Visualize the entire cell lineage tree of the nematode *C. elegans* (javascript)
- *Binary Trees* by L. Allison (http://www.allisons.org/ll/AlgDS/Tree/)

Categories: Data types | Trees (data structures) | Knowledge representation

---